Project 1: User Shell and Lottery Scheduling

Summary

- SiSH and Lottery Due April 27, 2016, 11:59:59PM
- To be completed in teams of two
- Includes two separate components, one inside Minix, one outside.

Now that you've all gotten familiar with Minix, we can start making more significant changes to Minix. We will start with the Shell and the process scheduler in Minix. In this assignment, we will implement a user shell that provides most functionalities present in common shells, then we will modify the Minix process scheduler to add Lottery scheduling, a cool scheduling algorithm that's not support by Minix.

Writing a Simple Shell: sish

While the goal is to change Minix, you can do this assignment in either Minix or on Linux. Building the shell in Minix is going to be more difficult than in Linux, so we will give an extra credit of 5% on your project if your shell works in Minix. However, my recommendation is to start with Linux to build and debug the core shell functionality, and only worry about the Minix port once you're satisfied with its performance in Linux.

The overall job of the shell program is to take input from the user, parse it, and perform the program or commands specified by the user. In addition to executing programs the user specifies, the shell also provides support for functionality such as input/output redirection, pipelining output from one program into another, and putting processes into the background.

The basic structure of the shell is essentially a long loop that parses user input. When running normally, the shell should always present the user with the prompt "`sish:>`" before each command is executed. In cases where STDIN is not a TTY (input redirection during invocation of your shell), no prompt should be displayed. To help you with basic I/O, you can make use of the `fgets()` function to read in lines from stdin. While parsing each line of input, you will need to support the following tokens/functionality:

- **Basic commands:** The user can type in basic commands which are simple alphanumeric tokens that specify a command, e.g. `ls`, `cat`. You should access the `PATH` environmental variable, and then search in the specified directories for the specified command. Once found, the command should be run in a new process, and the shell should block while waiting for the command to finish (see below for exceptions).
- **Command line arguments:** Each command can have command line arguments. You'll need to read in these arguments, and pass them along to the command you're running. Arguments are any text tokens that follow the command but are separated by white space.
- **Input/output redirection:** your shell will need to understand i/o redirection, as directed by special characters `<`, `>`, and `|`:
- `<`: The character '<' must be followed by a token that represents a file name. It means that the command before this character does not take input from the stdin of the shell, but instead reads its input from this file. '<' must follow the name of a command. Note that white space or white space characters between '<' and the text before and after it is optional. For example, `cat < file` and `cat<file` are equivalent.
- `>`: The '>' character does the opposite. `cmd > x` means that output from the command will be written out to file X instead of being written to stdout. The token before '>' must be a command, and the token after '>' is a filename. Again, the same rules regarding white spaces and white space characters apply as above.
- `|`: The '|' (i.e., pipe sign) allows multiple commands to be connected. The output of the command before the pipe sign must be connected to the input of the command after the pipe sign. This requires that there is a valid command both before and after the pipe. There can be multiple pipe signs on the command line. For example, your shell has to be able to process an input such as `cat f | sort | wc`. With this command, the output of the `cat` command is redirected to the input of sort, which in turn sends its output to the input of the `wc` program. The same rules as above applies to white space and white space characters.
- **Simplifications / clarifications**
- Only the first command in a line can read in input from a file
- Only the last command in a line can output to a file
- A single command can have both input and output redirection, *e.g.* `sort < infile > outfile`
- A single command cannot have two sources of input or two files for output, e.g. `cat | sort < file`, `cat x > file1 > file2` are not allowed.
- **Background processes:** The ampersand character '&' indicates that the command (or commands) specified should be executed in the background. The shell immediately displays a prompt to wait for the next line, even though commands on the previous line might not have exited yet. The '&' token may only appear as the last token of a line.
- **Exiting the shell:** the shell exits when the user input the 'exit' command (without quotes of course), or the ctrl-D or EOF character.
- **White space characters:** Any non-alphanumeric characters that are not one of the specialized characters described above is a white-space character. For example, invisible control characters like ^Q should be treated like blank space.
- **Error handling:** your shell should be robust to unexpected input. If there is unexpected input or errors that are generated by programs, the shell should output a message to standard out prefixed with "ERROR:". No input should be able to kill your shell or make it hang.

Additional Important Notes

- All shell output must go to standard out, however, it is okay if a command launched by your shell produces output on standard error.
- All errors (syntax, parsing, command not found) must be prefixed with "ERROR:" (less the quotes) and cannot exceed a single line.
- The source file for your shell must be named "shell.c" and must compile with this makefile.
- Your shell should produce this exact output (the error message on line 8 can be different as long as the ERROR: prefix is there) when using this input file via: ./shell < shell-input

Hints

- You should make use of the following system calls: `fork()`, `exec/execvp()`, `wait/waitpid()`, `dup2/dup()`, `pipe()` and `close()`
- Your input line will not be longer than 1024 characters
- You cannot use System, or Lex or Yacc.

Adding a Lottery Scheduler to Minix

The current Minix scheduler is broken into two parts. The selection of the next process to give the cpu to is done in kernel/proc.c, and uses 16 ready queues to control priority. The first process in the the first non-empty queue is chosen to run next, therefore if there is always a runnable process in queue 0, no other queue will have processes run from it. The scheduling of processes into different queues is done by the sched server (servers/sched/*). The current sched algorithm uses a simple algorithm where each process has a maximum priority, and the priority is decreased every time the process gets a turn, and increased periodically towards the maximum priority (using a timer). In the current system, a process can use the nice() system call to increase or decrease its maximum priority.

For this project you will be replacing the current sched algorithm with lottery scheduling for user processes. System processes use explicitly set maximum priorities which were intentionally chosen, so you should continue to use the default minix scheduling behavior for system processes. For all other processes, each process will start out with 5 tickets, and for non-system processes nice() will be overridden to add or subtract tickets instead of setting the maximum priority, with at most 100 tickets per process.

Every time a lottery is held, the scheduler should randomly select a ticket (by number), and then boost the priority of the process holding that ticket. The ticket selected should clearly be between 0 and one less than the total number of tickets. The scheduler should also regularly reduce the priority of processes. If a process wins many of the lotteries, they will have a lower (higher priority) average queue, whereas a process that wins fewer lotteries will have a higher (lower priority) average queue, and thus the amount of cpu time each process gets will be controlled.

Most of your code will be implemented in servers/sched/schedule.c, which contains the bulk of the sched server code, and this is also where your lotteries should be held. you may use random() to pick winning tickets, and the current date and time is a good value to seed the random number generator with. You may need to modify other files in servers/sched, e.g. for tracking tickets. Note that because lotteries must be held often, you should not simply count up all the tickets each time you hold a lottery. You will need to look through all the entry points of servers/sched/schedule.c to determine where and when you need to update the total number of tickets.

do_nice() in servers/sched/schedule.c implements updating process priority (and should be updated to adjust ticket counts for non-system processes). To handle nice values you will also need to update how the process managing server handles nice values in servers/pm/*, and how pm passes those values to sched (there are several files which will need to be edited).To make it simpler, you might update pm to simply pass along nice values directly to sched where possible, and then have sched either turn those values into ticket counts or maximum priorities depending on the process (right now pm turns the nice values into maximum priorities and then passes those to sched). you may also have to update the initialization of nice values in pm so the values for pm will match up with the number of tickets in sched.

It may also be useful to look at kernel/proc.c. This is where the low level scheduling occurs, and where the actual run queues are implemented and maintained. Looking at this code may help you get a better idea of how the pieces (pm,sched, and the run queues) fit together, such as when different calls are made to the sched server.

To get you started, in servers/sched/schedule.c, do_noquantum is called on behalf of processes which have just used up their turn (so this will be one place you have to handle rescheduling), and sched uses balance_queues() to perform operations periodically (with a timer).

Notes: Remember that system processes should continue to use the old scheduling behavior.

If you can keep the niceness in pm in sync with the number of tickets in sched, you will be able to use the top command to monitor the number of tickets various processes have and monitor how much cpu time processes are getting.

Submission Process

Your submission must be submitted prior to the deadline in order to be graded. Do a man turnin to find more info about the turnin program. To submit:

- Ensure your current working directory is a directory containing:
  - shell.c -- the file containing the code for your shell that runs on linix.
  - a file named patch that contains your changes to the Minix source. The patch must apply cleanly to a fresh source code tree and be run on your dev machine via:
    ```
    diff -ruNp minix_src_clean/ proj1/ > patch
    ```
  - an optional minix_shell.c which contains your shell ported to minix for extra credit
- Execute the turnin program:
  ```
  turnin proj1@cs170 patch shell.c
  ```

You can execute turnin as many times as required. The most recent submission prior to the deadline will be used for grading. You do not need to inform the TAs if you intend on using your two day extension for this project. Any submissions past the deadline will be assumed to be using your 2-day extension.